# Parallel Non-blocking Deterministic Algorithm for Online Topic Modeling

Oleksandr Frei[1] and Murat Apishev[2]

[1] Schlumberger Information Solutions, oleksandr.frei@gmail.com
[2] Lomonosov Moscow State University, great-mel@yandex.ru

**Abstract.** In this paper we present a new asynchronous algorithm for learning additively regularized topic models and discuss the main architectural details of our implementation. The key property of the new algorithm is that it behaves in a fully deterministic fashion, which is typically hard to achieve in a non-blocking parallel implementation. The algorithm had been recently implemented in the BigARTM library (`http://bigartm.org`). Our new algorithm is compatible with all features previously introduced in BigARTM library, including multimodality, regularizers and scores calculation. Previous BigARTM version was already faster than the alternative open source topic modeling packages such as Vowpal Wabbit or Gensim. Yet, in our experiments we show that the new implementation spends even less time to achieve the same perplexity, has better CPU utilization and lower memory usage.

**Keywords:** probabilistic topic modeling, Probabilistic Latent Sematic Analysis, Latent Dirichlet Allocation, Additive Regularization of Topic Models, stochastic matrix factorization, EM-algorithm, online learning, asynchronous and parallel computing, BigARTM.

## 1 Introduction

Deterministic behavior is an important property for any algorithm, including those of a stochastic nature. For the end users of a software system run-to-run reproducibility is a must-have property, because this is what they expect based on their previous experience. Indeed, refreshing a web-pages or re-doing an operation tend to produce an identical result, regardless of how much software complexity is hidden behind the scenes. For the researches determinism is also important because it enables them to reproduce their old experiments and study impact of various parameters on the result. Finally, for the developers of the algorithm determinism allow to reproduce bugs and write simple unit-tests with well-defined results.

Determinism is particularly hard to achieve in concurrent implementations, because in a multi-threaded environment it might not be sufficient to just fix a random seed or an initial approximation. In this paper we present a deterministic modification of parallel non-blocking algorithm for online topic modeling, previously developed for BigARTM library. We implement new algorithm in

BigARTM and demonstrate that new version converges faster than previous algorithm in terms of perplexity, yet being more efficient in CPU and memory usage.

The rest of the paper is organized as follows. In section 2 we introduce basic notation used throughout this paper. In sections 3, 4, 5 we summarize Offline, Online and Asynchronous Online algorithms for learning ARTM models. In section 6 we compare the internal architecture of BigARTM library between versions v0.6 and v0.7. In section 7 we report results of our experiments on large datasets. In section 8 we discuss advantages, limitations and open problems of BigARTM.

## 2 Notation

Let $D$ denote a finite set (collection) of texts and $W$ denote a finite set (vocabulary) of all terms from these texts. Let $n_{dw}$ denote the number of occurrences of a term $w \in W$ in a document $d \in D$; $n_{dw}$ values form a sparse matrix of size $|W| \times |D|$, known as *bag-of-words* representation of the collection.

Given an $(n_{dw})$ matrix, an additively-regularized topic model (ARTM) finds two matrices: $\Phi = \{\phi_{wt}\}$ and $\Theta = \{\theta_{td}\}$, of sizes $|W| \times |T|$ and $|T| \times |D|$ respectively, where $|T|$ is a used-defined number of *topics* in the model. Matrices $\Phi$ and $\Theta$ provide a compressed representation of the $(n_{dw})$ matrix:

$$n_{dw} \approx n_d \sum_{t \in T} \phi_{wt}\theta_{td}, \text{ for all } d \in D, w \in W,$$

where $n_d = \sum_{w \in W} n_{dw}$ denotes the total number of terms in a document $d$.

To learn $\Phi$ and $\Theta$ from $(n_{dw})$ ARTM maximizes the log-likelihood, regularized via an additional penalty term $R(\Phi, \Theta)$:

$$\sum_{d \in D} \sum_{w \in W} n_{dw} \ln \sum_{t \in T} \phi_{wt}\theta_{td} + R(\Phi, \Theta) \;\to\; \max_{\Phi,\Theta}. \tag{1}$$

With no regularization ($R = 0$) it corresponds to PLSA [?]. Many Bayesian topic models can be considered as special cases of ARTM with different regularizers $R$, as shown in [?,?].

In [?] it is shown that the local maximum $(\Phi, \Theta)$ of the problem (1) satisfies

$$p_{tdw} = \operatorname*{norm}_{t \in T}\bigl(\phi_{wt}\theta_{td}\bigr); \tag{2}$$

$$\phi_{wt} = \operatorname*{norm}_{w \in W}\left(\sum_{d \in D} n_{dw}p_{tdw} + \phi_{wt}\frac{\partial R}{\partial \phi_{wt}}\right); \tag{3}$$

$$\theta_{td} = \operatorname*{norm}_{t \in T}\left(\sum_{w \in d} n_{dw}p_{tdw} + \theta_{td}\frac{\partial R}{\partial \theta_{td}}\right); \tag{4}$$

where operator $\operatorname*{norm}_{i \in I} x_i = \frac{\max\{x_i,0\}}{\sum\limits_{j \in I} \max\{x_j,0\}}$ transforms a vector $(x_i)_{i \in I}$ to a discrete distribution.

**Algorithm 1:** Offline algorithm

```
Input: collection $D$;
Output: matrix $\Phi = (\phi_{wt})$;
1 initialize $(\phi_{wt})$;
2 form batches $D := D_1 \sqcup D_2 \sqcup \dots \sqcup D_B$;
3 repeat
4     $(n_{wt}) := \sum\limits_{b=1,\dots,B} \sum\limits_{d \in D_b} \mathsf{ProcessDocument}(d, \Phi)$;
5     $(\phi_{wt}) := \underset{w \in W}{\operatorname{norm}} (n_{wt} + \phi_{wt} \frac{\partial R}{\partial \phi_{wt}})$;
6 until $(\phi_{wt})$ converges;
```

**Algorithm 2:** ProcessDocument$(d, \Phi)$

```
Input: document $d \in D$, matrix $\Phi = (\phi_{wt})$;
Output: matrix $(\tilde{n}_{wt})$, vector $\theta_{td}$;
1 initialize $\theta_{td} := \frac{1}{|T|}$ for all $t \in T$;
2 repeat
3     $p_{tdw} := \underset{t \in T}{\operatorname{norm}} \big(\phi_{wt} \theta_{td}\big)$ for all $w \in d$ and $t \in T$;
4     $\theta_{td} := \underset{t \in T}{\operatorname{norm}} \big(\sum_{w \in d} n_{dw} p_{tdw} + \theta_{td} \frac{\partial R}{\partial \theta_{td}}\big)$ for all $t \in T$;
5 until $\theta_d$ converges;
6 $\tilde{n}_{wt} := n_{dw} p_{tdw}$ for all $w \in d$ and $t \in T$;
```

Learning of $\Phi$ and $\Theta$ from (2)–(4) can be done by EM-algorithm, which starts from a random initial approximation and iterates E-step (2) and M-steps (3),(4) until convergence. In the sequel we discuss several variations of such EM-algorithm, which are all based on the above formulas but differ in the way how operations are ordered and grouped together.

## 3 Offline algorithm

The algorithm is given by Fig. 1 (Offline algorithm) and Fig. 2 (ProcessDocument).

Subroutine ProcessDocument$(d, \Phi)$ corresponds to equations (2) and (4) from the fixed-point solution of the ARTM optimization problem (1). ProcessDocument requires a $\Phi$ matrix and a vector $n_{dw}$ of term frequencies for a given document $d \in D$, and as a result it returns the topical distribution $\theta_{td}$ for the document, and a matrix $(\hat{n}_{wt})$ of size $|d| \times |T|$, where $|d|$ gives the number of distinct terms in the document. The ProcessDocument might be also useful as a separate routine which finds $\theta_{td}$ distribution for a new document, but in the Offline algorithm it is rather used as a building block in an iterative EM-algorithm that learns the $\Phi$ matrix.

The Offline algorithm performs multiple scans over the collection, calls ProcessDocument for each document $d \in D$ from the collection, and then aggregates

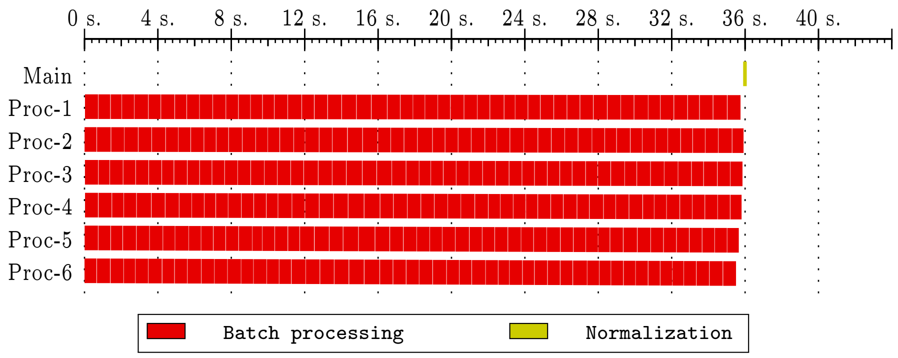

Fig. 1. Gantt chart for Offline algorithm

the resulting $(\hat{n}_{wt})$ matrices into the final $(n_{wt})$ matrix of size $|W| \times |T|$. After each scan it recalculates $\Phi$ matrix according to the equation (3).

Step 2, where we split collection $D$ into batches $(D_b)$, is not strictly necessary in the Offline algorithm, and it rather reflects an internal implementation detail. For performance reasons the outer loop over batches $b = 1, \dots, B$ is parallelized across multiple threads, and within each batch the inner loop $d \in D_b$ is executed in a single thread. Each batch is stored in a separate disk file on disk to allow out-of-core streaming of the collection. For typical collections it is reasonable to have around 1000 documents per batch, however for ultimate performance we encourage users to experiment with this parameter. Too small batches can cause disk IO overhead due to lots of small reads, while too large batches will result in bigger tasks that will not be distributed evenly across computation threads.

Note that $\theta_{td}$ values appear only within ProcessDocument subroutine. This leads to efficient memory usage because the implementation never stores the entire theta matrix at any given time. Instead, $\theta_{td}$ values are recalculated from scratch on every pass through the collection.

Fig. 1 illustrates a run-time behavior of the Offline algorithm. It shows a Gantt chart, where boxes correspond to the time spent in processing an individual batch. The final box, executed on the main thread, correspond to the time spent in the step 4 to normalize $n_{wt}$ values and produce a new $\Phi$ matrix.

## 4 Online algorithm

The Online algorithm improves the convergence rate of the Offline algorithm by re-calculating matrix $\Phi$ after every $\eta$ batches. To simplify the notation we introduce a trivial subroutine

$$\mathsf{ProcessBatches}(\{D_b\}, \Phi) = \sum_{D_b} \sum_{d \in D_b} \mathsf{ProcessDocument}(d, \Phi)$$

that aggregates the output of ProcessDocument across specific set of batches at a constant $\Phi$ matrix. The algorithm is then given by Fig. 3. Here the split of the collection $D := D_1 \sqcup D_2 \sqcup \dots \sqcup D_B$ into batches plays far more significant role than in the Offline algorithm, because different splitting algorithmically affects the result. At step 6 the new $n_{wt}^{i+1}$ values are calculated as a convex combination of the old values $n_{wt}^i$ and the value $\hat{n}_{wt}^i$ produced on the recent batches. Old counters $n_{wt}^i$ are discounted by a factor $(1 - \rho_i)$, which depends on the iteration number. Typical strategy is to use $\rho_i = (\tau_0 + i)^{-\kappa}$, where typical values for $\tau_0$ are between 64 and 1024, for $\kappa$ — between 0.5 and 0.7.

As in the Offline algorithm, the outer loop over batches $D_{\eta(i-1)+1}, \dots, D_{\eta i}$ is executed concurrently across multiple threads. The problem with this approach is that all threads have no useful work to do during steps 5-7 of the Online algorithm. The threads can not start processing the next batches because a new version of $\Phi$ matrix is not ready yet. As a result the CPU utilization stays low, and the run-time Gantt chart of the Online algorithm typically looks like in Fig. 3. The color indicate which version of the $p_{wt}^i$ matrix was used to process each batch (orange for even $i$, yellow for odd $i$). Blue box correspond to the time spend in merging $n_{wt}$ with $\hat{n}_{wt}$, Green box is, as before, the time spent to normalize $n_{wt}$ values and produce a new $p_{wt}$ matrix.

In the next section we present an asynchronous non-blocking modification of the online algorithm that results in better CPU utilization.

**Algorithm 3:** Online algorithm

```
Input: collection $D$, parameters $\eta, \tau_0, \kappa$;
Output: matrix $\Phi = (\phi_{wt})$;
1 form batches $D := D_1 \sqcup D_2 \sqcup \dots \sqcup D_B$;
2 initialize $(\phi_{wt}^0)$;
3 for all update $i = 1, \dots, \lfloor B/\eta \rfloor$
4     $(\hat{n}_{wt}^i) :=$ ProcessBatches$(\{D_{\eta(i-1)+1}, \dots, D_{\eta i}\}, \Phi^{i-1})$;
5     $\rho_i := (\tau_0 + i)^{-\kappa}$;
6     $(n_{wt}^i) := (1 - \rho_i) \cdot (n_{wt}^{i-1}) + \rho_i \cdot (\hat{n}_{wt}^i)$;
7     $(\phi_{wt}^i) := \operatorname{norm}_{w \in W}(n_{wt}^i + \phi_{wt}^{i-1} \frac{\partial R}{\partial \phi_{wt}})$;
```

## 5 Asynchronous online algorithm

Asynchronous online algorithm is based on two new routines, AsyncProcess-Batches and Await. The first one is equivalent to ProcessBatches, except that it just queues the task for an asynchronous execution and returns immediately. Its output is a future object (for example, an std::future from C++11 standard), which can be later passed to Await in order to get the actual result, e.g. in our

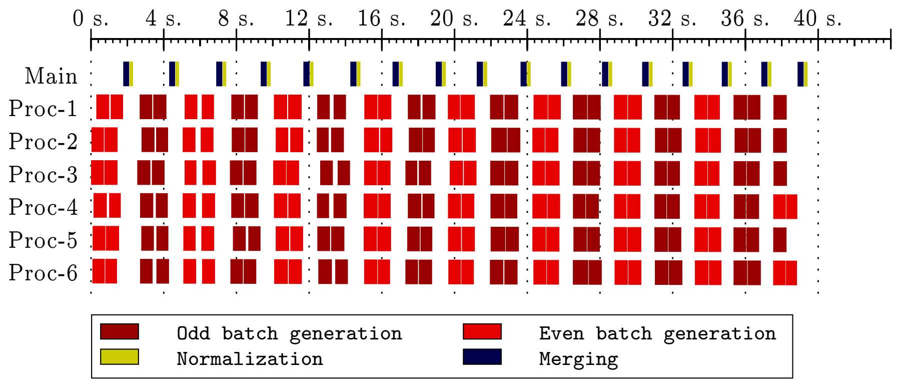

Fig. 2. Gantt chart for new online algorithm

case the $\hat{n}_{wt}$ values. In between calls to AsyncProcessBatches and Await the algorithm can perform some other useful work, while the background threads are calculating the $\hat{n}_{wt}$ values.

The resulting algorithm is given by Fig. 4. To calculate $\hat{n}_{wt}^{i+1}$ it uses $\Phi^{i-1}$ matrix, which is one generation older than $\Phi^{i}$ matrix used by the conventional Online algorithm 3. This adds an extra "offset" between the moment when $\Phi$ matrix is calculated and the moment when it is used, and as a result gives the algorithm additional flexibility to distribute more payload to computation threads. Steps 3 and 5 of the algorithm are just technical tricks to implement the "offset" idea.

Adding an offset should negatively impact the convergence of the asynchronous algorithm 4 comparing to the conventional algorithm **??**. For example, in AsyncProcessBatches the initial matrix $\Phi^0$ is used twice, and the two last matrices $\Phi^{\lfloor B/\eta \rfloor - 1}$ and $\Phi^{\lfloor B/\eta \rfloor}$ will not be used at all. One the other hand the asynchronous algorithm gives better CPU utilization, as clearly shown by the Gantt chart from Fig. 3.

This tradeoff convergence and CPU utilization is evaluated in the experiments from section 7.

## 6 Implementation

The challenging part for the implementation is to aggregate the $\hat{n}_{wt}$ matrices across multiple batches, given that they are processed in different threads. The way BigARTM solves this challenge was changed between versions v0.6 (see Fig. 4) and v0.7 (see Fig. 5). In the old architecture the $\hat{n}_{wt}$ matrices were stored in a queue, and then aggregated by a dedicated *Merger thread*. This often caused performance bottlenecks, particularly because of small batches or due to a small number of iterations in ProcessDocument's inner loop 2-5.

**Algorithm 4:** Asynchronous online algorithm

```
    **Input**: collection $D$, parameters $\eta, \tau_0, \kappa$;
    **Output**: matrix $\Phi = (\phi_{wt})$;
1   form batches $D := D_1 \sqcup D_2 \sqcup \dots \sqcup D_B$;
2   initialize $(\phi^0_{wt})$;
3   $F^1 :=$ AsyncProcessBatches$(\{D_1, \dots, D_\eta\}, \Phi^0)$;
4   **for all** *update* $i = 1, \dots, \lfloor B/\eta \rfloor$
5       **if** $i \neq \lfloor B/\eta \rfloor$ **then**
6           $F^{i+1} :=$ AsyncProcessBatches$(\{D_{\eta i+1}, \dots, D_{\eta i+\eta}\}, \Phi^{i-1})$;
7       $\hat{n}^i_{wt} :=$ Await$(F^i)$;
8       $\rho_i := (\tau_0 + i)^{-\kappa}$;
9       $(n^i_{wt}) := (1 - \rho_i) \cdot (n^{i-1}_{wt}) + \rho_i \cdot (\hat{n}^i_{wt})$;
10      $(\phi^i_{wt}) := \underset{w \in W}{\text{norm}}(n^i_{wt} + \phi^{i-1}_{wt} \frac{\partial R}{\partial \phi_{wt}})$;
```

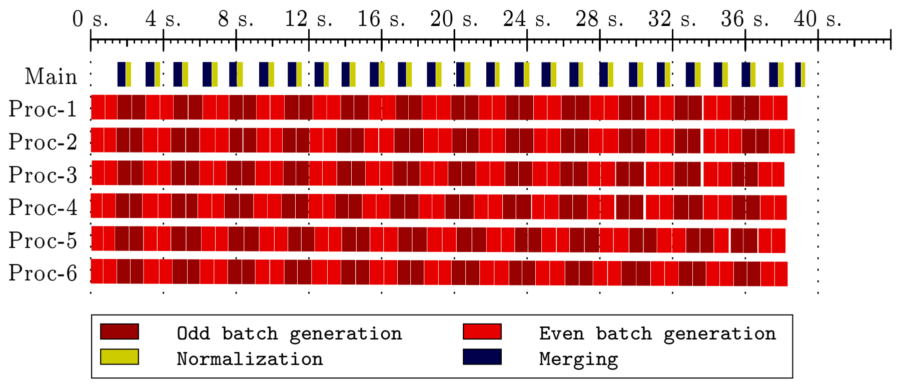

Fig. 3. Gantt chart for async online algorithm

In the new architecture we removed Merger thread, and $\hat{n}_{wt}$ are written directly into the final $n_{wt}$ matrix concurrently from all processor threads. To synchronize the write access we require that no threads simultaneously update the same row in $\hat{n}_{wt}$ matrix, yet the data for distinct words can be written in parallel. This is enforced by spin locks $l_w$, one per each word in the dictionary $W$. At the end of ProcessDocument we loop through all $w \in d$, acquire the corresponding lock $l_w$, append $\hat{n}_{wt}$ to $n_{wt}$ and release the lock. This approach is similar to [**?**], where the same pattern is used to update a shared stated in a distributed topic modeling architecture.

In our new architecture we also removed *DataLoader* thread, which previously was loading batches from disk. Now this happens directly from processor thread, which simplified the architecture without sacrificing performance.

In addition, we provided a cleaner API so now the users may use simple FitOffline, FitOnline methods to learn the model, and Transform to apply the

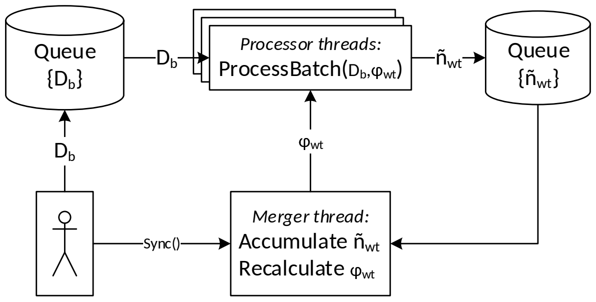

Fig. 4. Diagram of parallelization components

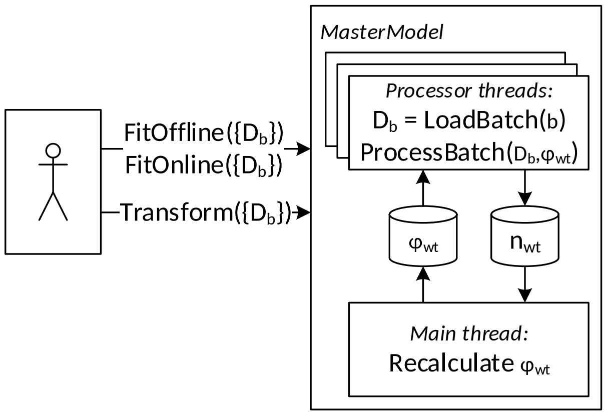

Fig. 5. Diagram of parallelization components (new architecture)

model to the data. Previously the users had to interact with low-level building blocks, such as ProcessBatches routine.

## 7 Experiments

Fig. 6 (*Wikipedia* dataset) and Fig. 7 (*Pubmed* dataset) show the perplexity as a function of the time spend by the following four algorithms: Offline algorithm (Fig. 1), Online algorithm (Fig. 3), asynchronous online algorithm (Fig. 4), and the old non-deterministic online algorithm from BigARTM v0.6, which is further described in Appendix 8.

Each point in the figures corresponds to a moment when the library finished a complete scan of the collection. Each algorithm was given a fixed budget (60 minutes). Lower perplexity means better result.

## 8 Conclusions

TBD

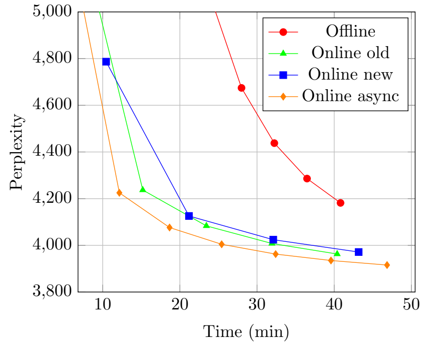

Fig. 6. 16 cores with hyper-threading, 30 minutes limit

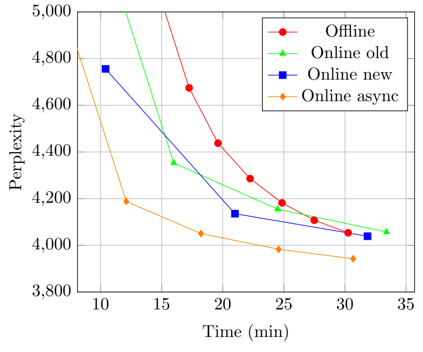

Fig. 7. 12 physical cores, 40 minutes limit

**Acknowledgements.** TBD

## References

TBD

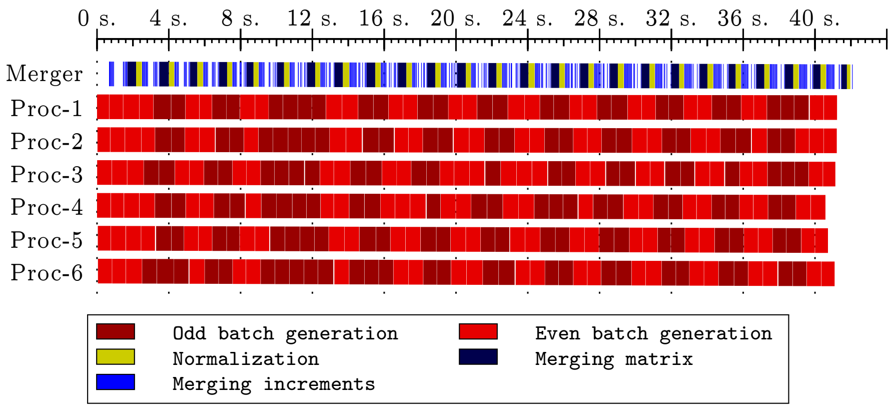

Fig. 8. Gantt chart for old online algorithm (TBD: include merger thread here)

## Appendix A

TBD: describe non-deterministic online algorithm from BigARTM $v0.6$ and its gantt chart 8.